# SystemVerilog アサーション入門(後編)

## 静的に解析するプロパティ検証にも利用可能

赤星博輝

本稿では，あらかじめ内部信号のふるまいを定義し，回路がそのとおりに動作しているかどうかをシミュレータに自動監視させるSystemVerilogアサーションについて解説する．前編(本誌2005年9月号，pp.83-94)ではアサーションを使う利点や基本構文，繰り返し記述，シーケンスなどについて解説した．後編ではアサーションを利用する際の注意点やハフマン符号デコーダの検証記述例などを紹介する．（編集部）

### ● 信号のふるまいに対応した関数が定義されている

アサーションでよく使われる関数のうちのいくつかを説明します(**表1**).

**1）信号変化を調べる関数($rose，$fell，$stable)**

信号の変化状態を調べる関数として，$rose，$fell，$stableの三つがあります．**図1**で示すように，信号aの値が‘0’から次のサイクルで‘1’になったときに成立するのが$rose(a)，信号aの値が‘1’から次のサイクルで‘0’になったときに成立するのが$fell(a)，信号aの値が前のサイクルと同じ場合に成立するのが$stable(a)です．

$rose，$fellはよく使う関数です．例えば，「要求reqを出すと，2～4サイクル以降いつかackが返ってくる」というプロパティについて，二つの記述を作成しました．CHECK1はreqが条件ですが，CHECK2は$rose(req)が条件という点が異なります．この差を**図2**で見てみます．

```
property CHECK1;
  @(posedge clk)   req |-> ##[*2:4] ack;
endproperty
```

**表1　アサーションで使われる関数の例**

| 関　数 | 動　作 |
|---|---|
| $rose | 立ち上がりの判定 |
| $fell | 立ち下がりの判定 |
| $statble | 無変化の判定 |
| $past | 過去の信号値を参照 |
| $onehot | 信号がワンホットか判定 |
| $onehot0 | 信号がワンホットもしくはALL0か判定 |
| $countones | 2進数表記で‘1’の数を返す |
| $isunknown | 信号にX，Zを含むかを判定 |
| $sampled | アサーションで使用する値を取得 |

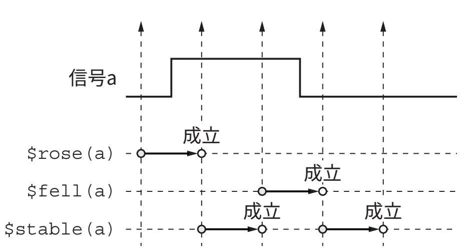



**図1　関数$rose，$fell，$stableの動作**

$rose，$fell，$stableはそれぞれ，信号の立ち上がり，立ち下がり，無変化の判定に使用する．

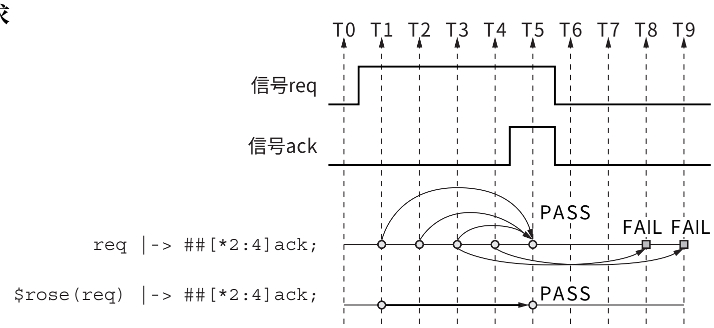



**図2　立ち上がり判定$roseを使うと便利な場合**

$roseを使わないで記述すると，reqが‘1’になった回数だけ応答するackが必要になる．$roseを使うことで，立ち上がりに対して1回だけackが必要となる．



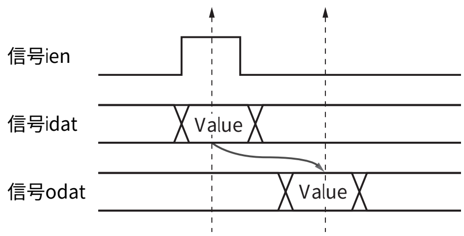



**図3　$pastを使う必要がある例**
「ienが'1'ならば，そのときのidatと次のサイクルのodatが等しい」というアサーションを書く場合，$pastが必要になる．

```
property CHECK2;
  @(posedge clk)   $rose(req) |->
                          ##[*2:4] ack;
endproperty
```

CHECK1はreqが'1'となった場合にチェックを開始するので，reqが'1'である5サイクルの間(T1～T5)に五つのアサーションが起動されます．このとき，前半の3サイクル(T1～T3)で起動されたCHECK1のアサーションはT5で成立しますが，後半の2サイクル(T4～T5)で起動されたCHECK1のアサーションはT8，T9で非成立となります．

これに対してCHECK2では$rose(req)の場合にチェックを開始するので，T1で起動されるだけになります．

**2）過去の値を見るには$past**

プロパティによっては，過去の信号の値と現在の信号の値を比較したいことがあります．例えば，「1ビットの信号ienが'1'ならば，このときの4ビットの信号idatの値と次のサイクルのodatの値が同じである」というプロパティです(**図3**)．これまでの記法では，異なった時刻の信号値を比較することができませんが，$pastを使うことでこの問題を解消できます．$pastの使いかたを以下に示します．

- $past(x)：信号xの1サイクル前の値
- $past(x, 5)：信号xの5サイクル前の値

この$pastを使って先ほどのプロパティを記述してみると，以下のように書けます．

```
ien |=>  odat == $past(idat)
```

**3）内部変数を利用したアサーション**

SystemVerilogの特徴として，アサーションを記述するときに内部変数を使えることが挙げられます．この機能がアサーションの記述力を大幅に向上させています．

**リスト1　内部変数を利用したアサーション**

```
property example;
  内部変数定義;
  クロック指定;
  disable iff定義;
  プロパティ定義;
endproperty
```

(**a**) 内部変数定義の位置

```
property pDAT;
  reg[3:0] v_dat;
  @(posedge clk)
  ($rose(a), v_dat=idat) |=> odat == v_dat;
endproperty
```

(**b**) 内部変数定義を使った記述例1

```
property pDAT2;
  reg[3:0] v_dat;
  @(posedge clk)
  (ien, v_dat=idat) |-> ##[2:3] oen &&(v_dat==odat);
endproperty
```

(**c**) 内部変数定義を使った記述例2

まず，プロパティで内部変数を定義する場所は，クロック指定の前になります(**リスト1(a)**)．内部変数の定義方法はVerilog HDLの場合と同じで，Verilog HDLおよびSystemVerilogのデータ型を使用できます．

内部変数に値を代入するためには，,を使って記述します．「1ビットの信号aが'1'ならば，そのときの信号xの値を内部変数vに代入する」というのは，以下のように記述できます．

```
(a, v=x)
```

その内部変数の値を参照するには，通常の信号と同じように変数名を使用します．

$pastのところで示した「1ビットの信号aが立ち上がりならば，このときの4ビットの信号idatの値と次のサイクルのodatの値が同じである」というプロパティに対して，内部変数を使った記述例を**リスト1(b)**に示します．

$pastだけでは記述しにくい場合でも，内部変数を使うと記述しやすくなることがあります．例えば，「1ビットの信号ienが'1'ならば，2～3サイクル後に1ビットの信号oenが'1'になり，そのとき(oenが'1'になったとき)のodatの値はienが'1'のときのidatの値である」というプロパティは**リスト1(c)**のように書けます．

**● 信号値の観測では実行順序に注意**

カウンタの例で実験してみます．例えば，「信号count

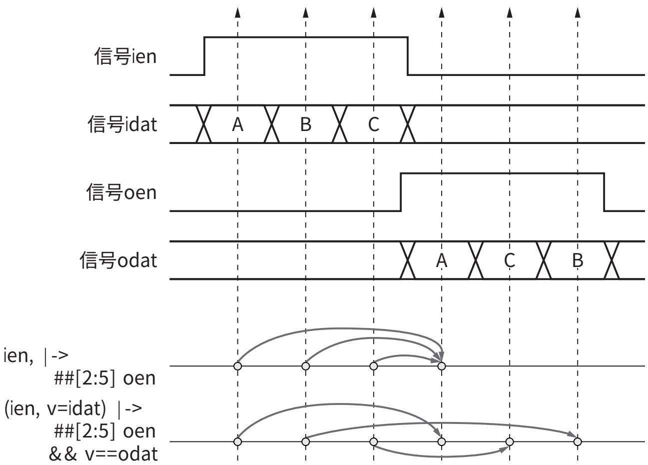



**図4　一見正しいアサーション**

`ien |-> ##[2:5] oen`は一見正しく見えるが，実は最初の`oen`でPASSしてしまう．パイプライン処理の場合は，データの値などを利用することで正しく検証できる．

の値が4未満である」というプロパティに違反した場合の信号countの値を出力してみます．

```
assert property (@(posedge clk) count
    <4 ) else $display("NG: %d", count);
```

この結果は，驚くべきことにNG: 5になります．ふつうはcountが4で違反なので，NG: 4になると思われた方が多いのではないでしょうか．

これはSystemVerilogの実行順序の問題が関係しています．すなわち，最初にアサーションの値を確保し，回路のシミュレーションを行い，次に値の更新を行い，その後，アサーションの判定を行って，メッセージを出力するためです．アサーションで使っている値を観測するためには，$sampled(count)とする必要があります．以下のように記述することで，期待される動作になります．

```
assert property (@(posedge clk) count <4)
else $display("NG: %d", $sampled(count));
```

### ● ノンブロッキングのときチェックできない場合がある

アサーションを仕様どおりに記述しても，実はチェックできないという例があります．「信号ienが'1'ならば，2～5サイクル後に信号oenが'1'になる」という仕様を例に考えてみます．このアサーションは，以下のように書けます．

```
assert property (@(posedge clk) ien  |->
                              ##[2:5] oen);
```

しかし，このアサーションで正しくチェックできるかどうかは，実は定かではありません．この処理がブロッキング処理で，oenが返ってくるまで次のリクエスト（ienが'1'）が来ない場合は正しくチェックできます．しかし，ノンブロッキング処理で，oenが返る前に次のリクエストが来る場合は，次のような問題が生じます．**図4**に示すように，最初にoenが'1'になったところで，すべてのアサーションがPASSします．これでよい場合（アービタなど）もあるのですが，ienとoenの対応がとれていなければならない場合もあります．

こういった場合，なんらかの情報を使うことで対応をチェックする必要があります．例えばienと同時に入力された4ビットのデータidatがoenと同時にodatに出力される場合，**リスト2**のようにしてチェックすることができます．

**リスト2　ノンブロッキング処理への対策**

```
assert property (
  reg[3:0] v;
  @(posedge clk)
  (ien, v=idat) |-> ##[2:5] oen && v==odat;
);
```

### ● アサーションは静的なプロパティ検証にも使える

アサーションを書いてシミュレーションすることで，検証の自動化が可能になりますし，人手でチェックするよりも多くの場所を観測することができます．しかし，アサーションをそれだけで利用するのはもったいないと思います．アサーションを用いた検証には，動的なシミュレーションを用いる方法のほかに，形式的検証（フォーマル・ベリフィケーション）ツールの一種であるプロパティ検証ツール（モデル・チェッキング・ツールとも呼ばれる）を用いて静的に解析する方法があります（最近では，動的なシミュレーションと静的なプロパティ検証を組み合わせたEDAツールも出荷されている）．ここでは米国Synopsys社の「Magellan」を利用して，シミュレータとプロパティ検証ツールの違いを見てみます．

**図5(a)**のHDL記述にあるように，3ビットのアップ・カウンタが0～6を繰り返します．そして，検証するとき



```
always@(posedge clk or negedge reset_n) begin
  if ( !reset_n ) begin
                  count <= 3'd0;
  end
  else begin
    if ( load ) begin
                  count <= value;
    end
    else if ( count == 3'd6 ) begin
                  count <= 3'd0;
    end
    else begin
                  count <= count + 3'd1;
    end
  end
end
```

(**a**) カウンタの記述




(**b**) Magellanの結果

**図5　カウンタとプロパティ検証ツールの結果**

**値をロードする機能がない(もしくはあることを知らない)という前提でテストベンチを作成すると，このアサーションでFAILを出すことが難しい．プロパティ検証ツール では，そういった面も見逃すことはない．**

**リスト3　3ビット・アップ・カウンタのアサーション**

```
property p2_34(x);
  @(posedge clk)
  disable iff (!reset_n)
  (x==3'd2) |=> (x==3'd3) ##1 (x==3'd4);
endproperty
CHL1:assert property (p2_34(count));
```

(**a**)「xの値が2になれば，次のサイクルでxは3，その次のサイクルでxは4になる(ただし，リセット時を除く)」

```
property pUpLess6(x);
  @(posedge clk)
  disable iff (!reset_n)
  (x<6) |=> (($past(x)+1'b1) == x);
endproperty
CHL2: assert property (pUpLess6(count));
```

(**b**)「xの値が6未満ならば，次のサイクルでxはインクリメントされている(ただし，リセット時は除く)」

```
property pUpEq6(x);
  @(posedge clk)
  disable iff (!reset_n)
  (x==6) |=> (x==0);
endproperty
CHL3: assert property (pUpEq6(count));
```

(**c**)「xの値が6ならば，次のサイクルでxは0である」

```
property pNot7(x);
  @(posedge clk)
  x != 3'd7;
endproperty
CHL4:assert property (pNot7(count));
```

(**d**)「つねにxの値は7にならない」

に忘れられている機能として，任意の値をロードできるものとします．この回路のために，**リスト3**の四つのアサーションを用意しました．

ロードする機能があるため，**リスト3**の各アサーションは満たされないはずですが，アサーションをシミュレーションだけに適用した場合，この仕様違反を発見できるかどうかはテストベンチの作りかたに依存します．とくに，上記のように忘れられている機能は，テストベンチでカバーされることはまれだと思います．

一方，プロパティ検証ツールは，テストベンチを利用せずにHDL記述とアサーションを使って数学的に正しいかどうかを判定する強力なツールです．今回の設計に適用してみると，**図5(b)**に示したように四つのアサーションがすべて“Falsified”となり，アサーションに対して設計違反があることを示しています．これだけだとどこにバグがあるのかを見つけるのがたいへんなのですが，プロパティ検証ツールは違反がどのようにして発生したのかを示す波形を表示してくれます．

プロパティ検証ツールで波形を作成して表示した例が，**図6**です．例えばCHL2のアサーションの違反は，ロードする機能を使って最初に5をロードし，次のサイクルで4をロードした場合に起きることがわかります．この例では，ロードする機能を取り除いてから，再度，プロパティ検証ツールを使って検証すると，すべてのアサーションが証明され，仕様どおりの設計になっていることが示されました．テストベンチを使用しないため，シミュレーションだけでは見逃してしまうパターンについても，形式的検証ツールはアサーション違反を発見できます．

アサーションを書くことには労力がかかります．しかし，そのアサーションはシミュレーションだけでなく，静的な

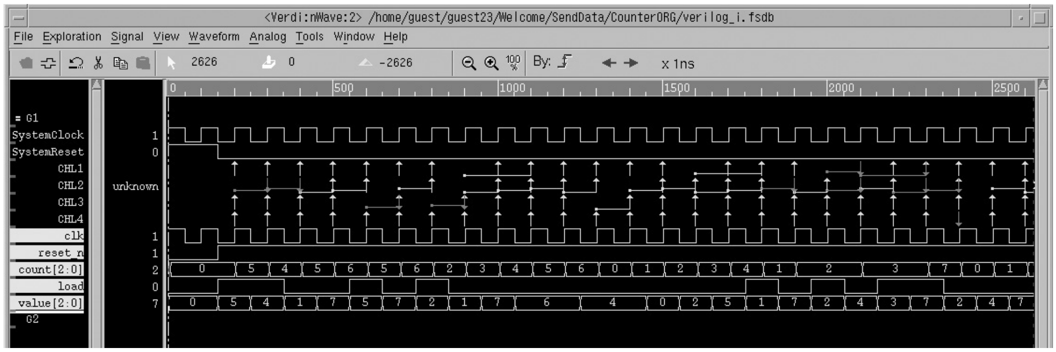



**図6**
**プロパティ検証ツールのエラー・パターンの生成**
**プロパティ検証ツールでは，FAILする場合の入力パターンを生成することができる．CHL2では，あるサイクルにおいてloadが1でvalueが5，次のサイクルではloadが1でvalueが4というパターンのとき，FAILする．**

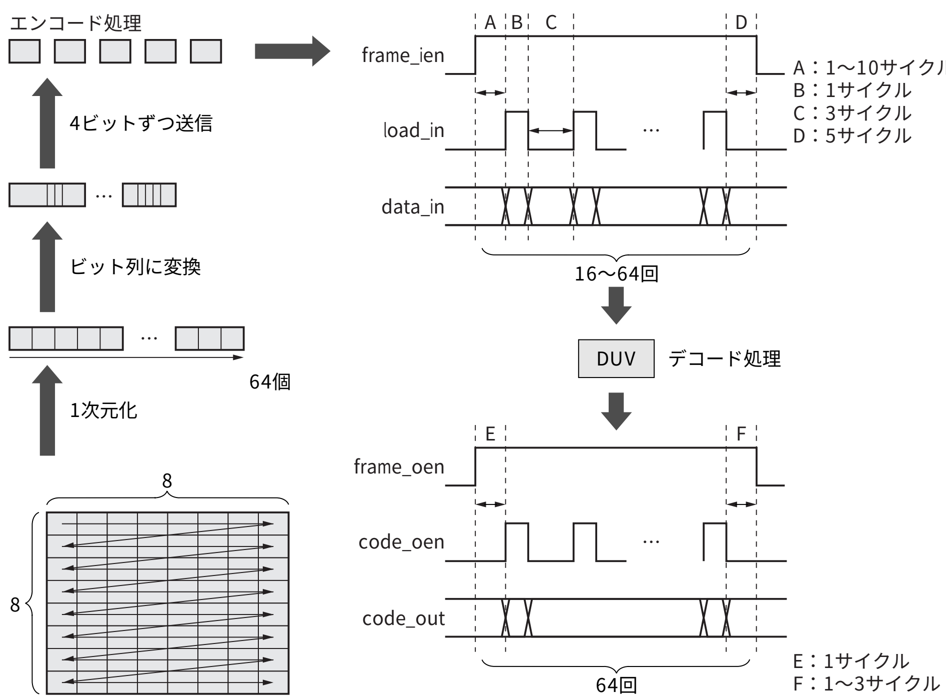



**図7**
**ハフマン符号のエンコード/デコード処理**
**検証対象となる回路はデコード処理部だけだが，テストベンチでエンコード部分もいっしょにモデリングすることにより，検証効率が向上する．**

プロパティ検証にも利用できます．このようなツールを利用することで，シミュレーションだけでは検出しにくいバグを見つけることができるのです．

### ● ハフマン符号デコーダのバグを検出

最後に，もう少し複雑な設計に対する検証について見ていきます．ここで検証するのは，8×8（ピクセル）の画像データを復号するハフマン符号のデコーダです．ハフマン符号では，よく出現する記号には短いビット列を，あまり出現しない記号には長いビット列を割り当てます．こうすることで，符号化する前のデータと比較してデータ・サイズを減らします．記号は0〜7の3ビットのデータを割り当てていますが，それを**表2**のように1ビット〜4ビットのビット列に割り当てます．デコーダは，ビット列からもとの記号に戻します．

**表2　ハフマン符号の割り当て**

| 元のコード | ハフマン符号 | |
|---|---|---|
| | ビット列 | ビット長 |
| 0（＝3'b000） | 1'b1 | 1 |
| 1（＝3'b001） | 3'b011 | 3 |
| 2（＝3'b010） | 4'b0101 | 4 |
| 3（＝3'b011） | 4'b0100 | 4 |
| 4（＝3'b100） | 4'b0011 | 4 |
| 5（＝3'b101） | 4'b0010 | 4 |
| 6（＝3'b110） | 4'b0001 | 4 |
| 7（＝3'b111） | 4'b0000 | 4 |

今回の回路は，8×8のデータを受けてデコードした結果を出力する必要があります．エンコードとデコードの流れを**図7**で説明します．まずは，8×8の2次元データを64



**リスト4 インターフェースをチェックするアサーションの例**

```
property IF_I1;
  @(posedge clk)
  disable iff (!reset_n)
  rose(frame_ien) |-> !load_in[*1:10] ##1 (load_in ##1 !load_in[*3])[*16:64]; [*2];fell(frame_ien);
endproperty
assert property (IF_I1);

property IF_O1;
  @(posedge clk)
  disable iff (!reset_n)
  rose(frame_oen) |-> !code_oen ##1 code_oen[->64]; [*0:2];fell(frame_oen);
endproperty
assert property (IF_O1);
```

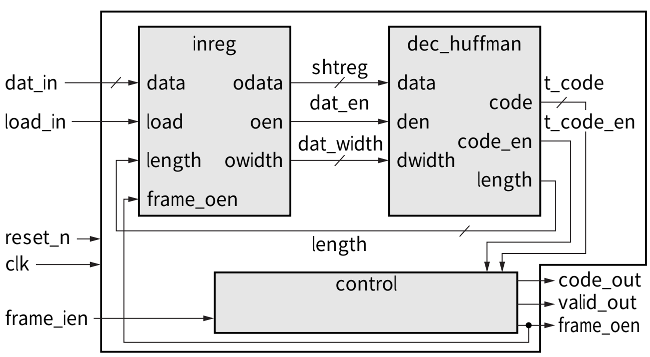



**図8 ハフマン・デコーダの内部構造**

**アサーションをモジュール内部に適用するには，内部の構造をはっきりさせる必要がある．機能や構造がクリアになっていない回路では，何をチェックするべきかがはっきりしないことが多く，その場合，アサーションを書くことができない．**

個の1次元データに変換し，**表2**をもとに各コードを対応するビット列に変換します．この変換後のビット長は，与えられたデータによって変わる点に注意が必要です．

このビット列を4ビットずつデコーダ側に送信します．このデータを送る際には，フレーム信号を立ち上げてからデータを送信します．また，1度データを送信して次にデータを送信するまで，3クロックのウェイトが挿入されます．すべてのデータを送信した場合は，送信して3クロックたってからフレーム信号を‘0’にします．また，送信するデータ・サイズが4の倍数でない場合，最後のデータは「0詰め」して送信します．ここで規定された条件は，**図7**にA～Fとして示しています．

そして，デコーダが出力する場合にも出力用のフレーム信号を出力し，その後，データを出力します．こちらは連続してデータを出力してもよい仕様とします．そして，64個のデータを出力してから1～3クロック後にフレーム信号を‘0’にします．

## ● インターフェース部をアサーションでチェック

インターフェースの部分は設計の境界面であると同時に，設計者や設計グループの境界面であることが多く，どうしてもミスが発生しやすい箇所です．そこで，アサーションによる検証が重要になります．また，標準化されたインターフェースであれば，すでに作成済みのアサーション記述を再利用できる場合もあります(**リスト4**)．

ここでは**図7**で示したAからFの条件について，System Verilogアサーションを使って検証してみます．

現在のLSI設計では，内部の値を観測するのはデバッグの段階がほとんどで，バグがまだ出ていない段階で内部の値を観測してシミュレーションすることはあまりないと思います．しかし，外部にバグが出てきたときにその原因がどこにあるかを突き止めるのはたいへんですし，バグの影響がかならず外部に出てくるとは限りません．そのため，モジュール内部にアサーションを適用してバグの位置を早期に特定したり，バグを発見する可能性を増やします．

今回のハフマン符号デコーダの内部は，**図8**のように三つに分けて構成されています．inregでは4ビットずつ入力されるデータを保存し，そのデータをdec_huffmanに渡します．dec_huffmanではinregで保存されたデータからデコード処理を行い，出力としてデコードしたコードをcodeに出力すると同時に，コードが有効かどうかを示すcode_en信号を‘1’にします．このとき，出力したコードのビット長をinregに返します．inregではそのビット長分はデコードが終了したということで，そのビット幅だけシフトしてデータを捨て，また有効なデータをdec_huffmanに出力します．controlはステート・マシンを持ち，データが正しくデコードできているかどうかを監視するために，信号countにそのフレームで出力されたデータをカウントします．

まず，内部のモジュール間を結ぶ配線を利用したアサー

| load | length | owidth |
|---|---|---|
| 1'b0 | 3'd0 | owidth |
| | 3'd1 | owidth－1 |
| | 3'd3 | owidth－3 |
| | 3'd4 | owidth－4 |
| 1'b1 | 3'd0 | owidth＋4 |
| | 3'd1 | owidth＋3 |
| | 3'd3 | owidth＋1 |
| | 3'd4 | owidth |

```
property noload_len0_width =
  always {!load&&length==3'd0} |=>{owidth==prev(owidth)}
  abort (!reset_n || !frame_oen);
assert noload_len0_width;
```

```
property load_len0_width =
  always {load&&length==3'd0} |=>{owidth==prev(owidth)+4}
  abort (!reset_n || !frame_oen);
assert load_len0_width;
```

（**a**）owidth の動作

| remain | odata（レジスタ） |
|---|---|
| 0 | { idat[3:0], 3'd0} |
| 1 | {s_dat[6], idat[3:0], 2'd0} |
| 2 | {s_dat[6:5], idat[3:0], 1'd0} |
| 3 | {s_dat[6:4], idat[3:0] } |

```
property remain0 =
  always { remain==0 } |-> {t_dat == {idat[3:0], 3'd0}}
  abort (!reset_n);
assert   remain0;
property odata_remain =
  always {! (!reset_n || !frame_oen)}|=>{odata == prev(t_dat)}
  abort (!reset_n);
assert odata_remain;
```

（**b**）odata の動作

**図9　モジュールinregの仕様の一部とアサーション**
**仕様書があれば，それをもとにアサーションを作成できる．仕様書がなければ，アサーションの作成は不毛な作業になる．**

ションについて検討しました．ここでは以下の三つの条件を抜き出しました．

- 「新しいデータを受信した場合（load_in==1）には，inregから出力されるビット幅は次のサイクルのほうが大きくなる（dat_width > $past(dat_width)）」
- 「デコードされたビット幅は，inregから出力されるビット幅以下である」
- 「inregから出力されるビット幅が0ならば，dec_huffmanで出力されるt_code，t_code_en，lengthは‘0’である」

これらをSystemVerilogアサーションで記述したものを**リスト5**に示します．

また，内部のモジュールについてもアサーションを記述しました．**図9**にあるように，inregの仕様からアサーションを記述できることがわかります．これもシミュレーションではFAILにならなかったのですが，プロパティ検証ツールではFAILになっていました．

## ● 仕様が明示されていなければアサーションの効果は低い

HDLのコードを記述する前に仕様書をどこまできちんと作るのかについてはさまざまな流儀があるのですが，アサーションを利用して検証する場合は仕様の情報が必須になります．これがないと，本来は疑ってかかるべきHDLをベースにアサーション記述を作成することになってしまいます．これでバグを検出できるでしょうか？

**リスト5　内部のインターフェースをチェックするアサーション**

```
property load_increase_width;
  @(posedge clk)
  disable iff (!reset_n)
  load_in |=> dat_width>$past(dat_width);
endproperty

assert property (load_increase_width);

property code_length;
  @(posedge clk)
  (length <= dat_width);
endproperty
assert property (code_length);

property nodata_nocode;
  @(posedge clk)
  dat_en ==0  |-> t_code==3'd0&&!t_code_en&&length
                                                  ==0;
endproperty
assert  property (nodata_nocode);
```

これではまちがっているかどうか判断できません．仕様の情報がない場合，アサーションを用いた検証にはあまり効果がないということを頭に入れておいてください．

**あかぼし・ひろき**
**(株)ロジック・リサーチ**

**<著者プロフィール>**
**赤星博輝．ロジック・リサーチに所属しており，ハードウェアからソフトウェアまで担当するなんでも屋．もともとはEDAツールの開発を行っていたが，現在はハードウェア設計から徐々に検証技術に移っているところ．今後，どのようなしごとをするのかは，世の中の進みかたしだいです．**